

# Multi-tasking

תהליכי רקע

# ריבוי משימות - Multi tasking

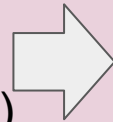
- למחשב עם ליבה בודדת אין אפשרות לבצע במקביל מספר תהליכים.
- הוא ממתג את זמן המעבד בין מספר תהליכים - אשליה.
- ג'אווה מספקת תמיכה בריבוי המשימות במספר דרכים.
- יצירת תהליך רקע על ידי ירושה מ Thread או Runnable
- נראה כיצד "לחכות" מתהליך אחד לאחר
- כיצד להגדיר תהליך שאינו עצמאי (daemon).
- כיצד להימנע מ Race conditions

# ירושה מ Thread

- תכנית רצה בתהליך ראשי שניתן לה על ידי המערכת
- תהליך נוסף ע"י ירושה מן המחלקה Thread כתיבת הקוד ב run, יצירת instance שלה וקריאה ל start עליו.
- מערכת ההפעלה תייצר Thread נוסף ותריץ שם את הקוד.
- פונקציה סטטית של המחלקה Thread בשם sleep(time) אומרת לתהליך ללכת לישון לזמן שביקשנו.
- התכנית תסתיים כאשר כל התהליכים הרגילים שלה יסתיימו.

# דוגמה לירושה מ Thread

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException ex)
            {
                ex.printStackTrace();
            }
            System.out.println("Background
Thread");
        }
    }
}
```



```
public static void main(String[] args) {
    new MyThread().start(); //no need for ref
    wont die untill the end
    for(int j=0;j<10;j++)
    {
        try
        {
            Thread.sleep(100);
        }
        catch (InterruptedException ex) //אם
        מישהו מהמערכת מעיר אותו בכוח מסיבה מסויימת
        {
            ex.printStackTrace();
        }
        System.out.println("Main Thread");
    }
}
```

הסיבה  
להמתנה היא  
לתת למעבד  
זמן למיתוג  
בין התהליכים

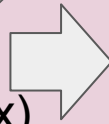
# מימוש Runnable

- אם נרצה מחלקה שתהיה יותר מאשר רק תהליך רקע, בעייתי שכן הירושה היא ממחלקה אחת בלבד.
- כפתרון ניתן לממש את הממשק של Runnable ולהעביר instance של המחלקה הזו ל constructor של Thread.
- ניתן גם לתת שם ל Thread הזה על ידי שימוש ב constructor שמקבל עוד ארגומנט והוא השם של התהליך הזה.
- השם של התהליך הראשי הוא main של תהליך נוסף יהיה Thread 1, Thread 0 וכך הלאה.

# דוגמה למימוש Runnable

```
class Job implements Runnable
{
    public void run()
    {
        for(int i=0;i<10;i++)
        try
        {
            Thread.sleep(100);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
    System.out.println("background");
}
```

נוכל לרשת מכל  
מחלקה שנרצה  
ולכתוב  
פונקציות  
נוספות



```
public static void main(String[] args) {
    new Thread(new Job(),"My
    Job").start();
    for(int i=0;i<10;i++) {
        try
        {
            Thread.sleep(100);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        System.out.println("main");
    }
}
```

# Daemon thread

- Thread שמוגדר כ daemon נחשב לתהליך שאינו עצמאי ותלוי בתהליכים אחרים.
- הוא לא יחזיק את האפליקציה בחיים.
- תכנית תסתיים כאשר כל התהליכים שאינם daemon יסתיימו.
- כאשר תהליך שמוגדר כ daemon עדיין לא הסתיים והתהליך הראשי הסתיים המערכת תסיים את ה thread הזה.
- כדי להגדיר thread כ daemon מפונקציה `setDaemon()` של ה thread instance את הערך `true`

# המתנה לתהליך join()

- הפונקציה join() אומרת לתהליך הנוכחי להמתין עד שהתהליך עליו אני מפעיל את ה join יסתיים.
- join(int) המקבלת זמן במילי - שניות ואומרת להמתין עד שהתהליך עליו קראנו אותה יסיים או עד אשר הזמן שמועבר יסתיים (מה שיקרה קודם).



# thread.join() example

```
class GoodMorningThread extends Thread {  
  
    @Override  
    public void run() {  
        super.run();  
        for (int i=0;i<5;i++) {  
            System.out.println("Good morning");  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

# thread.join() example

```
public static void main(String[] args) {
```

```
    Thread goodMorning = new GoodMorningThread();  
    goodMorning.start();
```

```
    try {  
        goodMorning.join();  
    } catch (InterruptedException ex) {  
        ex.printStackTrace();  
    }
```

```
    for (int i=0;i<5;i++) {  
        System.out.println("Good Evening");  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
    }
```

```
}
```

ההדפסה של ה"ערב  
טוב" לא תתחיל לפני  
שה"בוקר טוב"  
יסתיים

# Race Conditions

- מצב זה מתרחש כאשר תהליכים מקבילים ניגשים למשאב משותף.
- בעוד מקום אחד בודק את הערך יתרחש מיתוג לתהליך אחר ואז הערך שהתהליך הראשון בדק לא יהיה רלוונטי.
- הפתרון הוא להגדיר את הגישה למשאב המשותף כפעולה אטומית.
- פעולה אטומית מתרחשת בשלמותה ללא מיתוג של המעבד.

# Synchronized

- מגדירים אובייקט שכל מטרתו לשמש מנעול (ניתן שהוא יהיה סטטי או ניתן להעביר לתהליכים אותו אובייקט):

```
public static Object handleBalance=new Object();
```

- לפני שניגשים למשאב המשותף יוצרים בלוק מסונכרן לפי האובייקט:

```
synchronized(handleBalance.שם המחלקה) {
```

```
// everything that will be written here will be Atomic meaning
```

```
// the processor won't switch to another process in the middle
```

```
}
```

- כל הבלוק יתבצע בצורה אטומית - בשלמותו, ללא מיתוג של המערכת, כל תהליך אחר שירצה להיכנס יראה שהאובייקט נעול ויחכה בכניסה, כאשר הבלוק יסתיים תשתחרר הנעילה באובייקט ואז התהליך שמחכה יוכל להיכנס.

# wait() & notify()

- ניתן לבקש מתהליך אחד או יותר להמתין עד אשר יקבל הודעה להמשיך מתהליך אחר.
- סימון ההמתנה והאפשרות להמשיך נעשות באמצעות אובייקטים, הגישה לאובייקטים אלו צריכה להיות מסונכרנת.
- כאשר נרצה שתהליך מסוים ימתין, נקרא על האובייקט ל wait() (הפונקציות הללו מתקבלות מ Object).
- כאשר נרצה לשחרר את התהליך הממתין מתוך תהליך אחר נקרא על אותו אובייקט ל notify().
- אם נרצה לשחרר את כל התהליכים הממתינים נקרא ל notifyAll().